Les bases C

1. Structure de base d'un programme C

Un programme en C commence par la fonction `main()` qui est le point d'entrée du programme. La bibliothèque standard d'entrée/sortie (`stdio.h`) est souvent incluse pour utiliser des fonctions comme `printf` et `scanf`.

```c
#include <stdio.h>

int main() {
    // Code ici
    return 0;
}
```

- `#include` : Directive de préprocesseur utilisée pour inclure des fichiers d'en-tête.
- `int main()` : Fonction principale où commence l'exécution du programme.
- `return 0;` : Indique que le programme s'est terminé avec succès.

2. Types de données

Les types de données en C déterminent la nature des valeurs que les variables peuvent stocker.

- **Primitifs** :
  - `int` : Entier (4 octets)
  - `float` : Nombre à virgule flottante (4 octets)
  - `double` : Nombre à virgule flottante double précision (8 octets)
  - `char` : Caractère (1 octet)
  - `void` : Type vide, utilisé pour indiquer l'absence de type (ex: fonctions qui ne retournent rien)
- **Dérivés** :
  - **Tableaux** : Stocke des collections de données de même type. `int arr[10];`
  - **Pointeurs** : Stocke l'adresse mémoire d'une variable. `int *p;`
  - **Structures** : Regroupement de plusieurs variables sous un même nom. `struct Person { char name[50]; int age; };`
  - **Unions** : Comme les structures, mais les membres partagent la même zone mémoire.
- **Modificateurs de type** : Modifient la taille et le signe des types de base.
  - `short` : Réduit la taille d'un entier (généralement 2 octets).

- `long` : Augmente la taille d'un entier (généralement 8 octets).
- `signed` : Permet à une variable d'être positive ou négative.
- `unsigned` : La variable ne peut être que positive.

3. Constantes et variables

- **Variables** : Espaces mémoire réservés pour stocker des données. Ex : `int x = 10;`
- **Constantes** : Valeurs immuables. Ex : `const int MAX = 100;`
  - `#define` : Directive du préprocesseur pour définir des constantes. Ex : `#define PI 3.14`

4. Opérateurs

Les opérateurs permettent de manipuler les variables et les valeurs.

- **Arithmétiques** : `+` (addition), `-` (soustraction), `*` (multiplication), `/` (division), `%` (modulo).
- **Logiques** : `&&` (ET logique), `||` (OU logique), `!` (NON logique).
- **Relationnels** : `==` (égal), `!=` (différent), `<`, `>`, `<=`, `>=`.
- **Assignation** : `=` (affectation), `+=`, `-=`, `*=` (opérateurs combinés).
- **Bitwise** : `&` (ET bit à bit), `|` (OU bit à bit), `^` (OU exclusif), `~` (NON bit à bit), `<<` (décalage à gauche), `>>` (décalage à droite).

5. Structures de contrôle

Les structures de contrôle permettent de contrôler le flux d'exécution du programme.

- **Conditionnel** :

```
if (condition) {
    // Bloc exécuté si la condition est vraie
}
else if (autre_condition) {
    // Bloc exécuté si la première condition est fausse et la deuxième est vraie
}
else {
    // Bloc exécuté si toutes les conditions sont fausses
}
```

- **Switch** : Permet de sélectionner une des nombreuses branches d'exécution en fonction de la valeur d'une variable.

```
switch (variable) {
    case 1: // Instructions
            break;
    case 2: // Instructions
            break;
    default: // Instructions par défaut
}
```

- **Boucles** :
- `for` : Répète un bloc de code un nombre déterminé de fois.

```
for (int i = 0; i < 10; i++) {
    // Instructions
}
```

- `while` : Répète un bloc de code tant qu'une condition est vraie.

```
while (condition) {
    // Instructions
}
```

- `do...while` : Similaire à `while`, mais la condition est testée après l'exécution du bloc de code.

```
do {
    // Instructions
} while (condition);
```

6. Fonctions

Les fonctions permettent de regrouper du code réutilisable sous forme de blocs nommés.

- **Définition** : Une fonction peut recevoir des paramètres et retourner une valeur.

```
int add(int a, int b) {
    return a + b;
}
```

- **Prototypes** : Les déclarations de fonctions sont souvent placées avant le `main` pour informer le compilateur de leur existence.

```
int add(int, int);
```

7. Tableaux

Les tableaux sont des collections de données de même type stockées sous une seule variable.

- **Déclaration** : `int arr[10];` déclare un tableau de 10 entiers.
- **Accès aux éléments** : Utilisez l'indice pour accéder à un élément. `arr[0] = 5;`
- **Tableaux multidimensionnels** : `int mat[3][3];` déclare une matrice 3x3.

8. Pointeurs

Un pointeur est une variable qui stocke l'adresse mémoire d'une autre variable.

- **Déclaration** : `int *p;` déclare un pointeur vers un entier.
- **Assignation** : `p = &variable;` stocke l'adresse de `variable` dans `p`.
- **Déréférencement** : `int x = *p;` accède à la valeur pointée par `p`.

Les pointeurs sont également utilisés pour parcourir des tableaux, passer des paramètres par référence et gérer la mémoire dynamique.

9. Structures

Les structures permettent de regrouper plusieurs variables sous un même nom, souvent pour représenter des objets complexes.

- **Déclaration** :

```
`struct Person {     char name[50];     int age; };`
```

- **Accès aux membres** : Utilisez l'opérateur `.` pour accéder aux membres. Ex : `person.age = 30;`
- **Pointeurs sur structures** : Utilisez l'opérateur `->` pour accéder aux membres via un pointeur. Ex : `person_ptr->age = 30;`

10. Entrée/Sortie (I/O)

Les fonctions de la bibliothèque standard `stdio.h` permettent de gérer les entrées et sorties de données.

- **Lecture d'entrée** : `scanf()` permet de lire des données depuis l'entrée standard (clavier).

```
`scanf("%d", &variable);`
```

- **Affichage de sortie** : `printf()` permet d'afficher des données à l'écran.

```
`printf("La valeur est : %d\n", variable);`
```

11. Gestion de la mémoire

La gestion dynamique de la mémoire en C permet d'allouer et de libérer de la mémoire pendant l'exécution du programme.

- **Allocation dynamique** : Utilisez `malloc()` pour allouer de la mémoire sur le tas. `malloc` vient de `#include <stdlib.h>` il faut donc l'inclure.

```
`int *ptr = (int*) malloc(sizeof(int) * n);`
```

- **Libération de mémoire** : Utilisez `free()` pour libérer la mémoire allouée.

```
`free(ptr);`
```

12. Fichiers

Les fichiers permettent de stocker et de récupérer des données depuis un stockage permanent.

- **Ouverture de fichier** : `fopen()` ouvre un fichier pour la lecture, l'écriture ou les deux.

```
`FILE *file = fopen("nom_fichier.txt", "r");`
```

- **Lecture et écriture** : `fprintf()` et `fscanf()` permettent d'écrire et de lire dans des fichiers.

```
`fprintf(file, "Texte à écrire"); fscanf(file, "%d", &variable);`
```

- **Fermeture de fichier** : `fclose()` ferme un fichier pour libérer les ressources.

```
`fclose(file);`
```

13. Préprocesseur

Les directives de préprocesseur sont exécutées avant la compilation.

- **Inclusion de fichiers d'en-tête** : `#include` insère le contenu d'un fichier dans le programme.

```
`#include <stdio.h>`
```

- **Définition de constantes** : `#define` permet de définir des macros constantes ou des fonctions de macros.

```
`#define PI 3.14 #define SQUARE(x) (x * x)`
```

14. Gestion des erreurs

La gestion des erreurs en C implique souvent l'utilisation de codes de retour (`return`) et de tests de valeurs.

- **Retour d'erreur** : Utilisez `return EXIT_FAILURE;` pour indiquer un échec.
- **Gestion des pointeurs NULL** : Vérifiez si un pointeur est nul avant de l'utiliser.

```
`if (ptr == NULL) {     // Gestion de l'erreur }`
```

15. Bibliothèques standard

Les bibliothèques standard en C offrent une gamme de fonctions utiles :

- **`stdlib.h`** : Gestion de la mémoire, conversions de types, génération de nombres aléatoires. Ex : `malloc()` , `free()` , `atoi()` .
- **`stdio.h`** : Entrée/sortie standard. Ex : `printf()` , `scanf()` , `fopen()` , `fclose()` .
- **`string.h`** : Fonctions de manipulation de chaînes de caractères. Ex : `strcpy()` , `strlen()` , `strcmp()` .
- **`math.h`** : Fonctions mathématiques. Ex : `sqrt()` , `pow()` , `sin()` .

Les spécificateurs de format sont utilisés dans les fonctions de formatage en C, comme `printf()` et `scanf()` , pour indiquer le type de données à traiter. Voici un résumé des spécificateurs de format les plus courants :

1. Spécificateurs pour les Types Numériques

- `%d` **ou** `%i` : Entier décimal signé.
  - Exemple : `int x = 42; printf("%d", x);`
- `%u` : Entier décimal non signé.
  - Exemple : `unsigned int y = 42; printf("%u", y);`
- `%f` : Nombre à virgule flottante en notation décimale (float ou double).
  - Exemple : `float z = 3.14; printf("%f", z);`
- `%lf` : Nombre à virgule flottante en notation décimale pour un double.
  - Exemple : `double z = 3.14159; printf("%lf", z);`
- `%x` **ou** `%X` : Entier en hexadécimal non signé (en minuscule avec `%x`, en majuscule avec `%X`).
  - Exemple : `int n = 255; printf("%x", n);` // Affiche `ff`
- `%o` : Entier en octal non signé.
  - Exemple : `int n = 255; printf("%o", n);` // Affiche `377`
- `%e` **ou** `%E` : Nombre à virgule flottante en notation scientifique (en minuscule avec `%e`, en majuscule avec `%E`).
  - Exemple : `float z = 3.14; printf("%e", z);` // Affiche `3.140000e+00`
- `%g` **ou** `%G` : Utilise `%f` ou `%e` selon ce qui est plus court.
  - Exemple : `float z = 3.14; printf("%g", z);` // Affiche `3.14`

2. Spécificateurs pour les Caractères

- `%c` : Affiche un caractère unique.
  - Exemple : `char c = 'A'; printf("%c", c);`
- `%s` : Affiche une chaîne de caractères (un tableau de caractères terminé par un caractère nul `\0`).
  - Exemple : `char str[] = "Hello"; printf("%s", str);`

3. Spécificateurs pour les Pointeurs

- `%p` : Affiche une adresse mémoire sous forme hexadécimale.
  - Exemple : `int *ptr = &x; printf("%p", ptr);`

4. Spécificateurs pour la Largeur

- `%ld` : Entier long signé.
  - Exemple : `long l = 123456789L; printf("%ld", l);`
- `%lu` : Entier long non signé.
  - Exemple : `unsigned long ul = 123456789UL; printf("%lu", ul);`
- `%lld` : Entier long long signé (pour les très grands entiers).

    - Exemple : `long long ll = 1234567890123LL; printf("%lld", ll);`
- `%llu` : Entier long long non signé.
    - Exemple : `unsigned long long ull = 1234567890123ULL; printf("%llu", ull);`

5. Spécificateurs de Contrôle

- `%%` : Affiche le caractère `%`.
    - Exemple : `printf("Affiche un pourcentage : %%");`

6. Spécificateurs de Formatage Avancé

- `%*d` : Permet de spécifier la largeur d'un champ dynamiquement.
    - Exemple : `int width = 5; printf("%*d", width, 42);` // Affiche `42`
- `%.nf` : Contrôle le nombre de décimales affichées pour un flottant.
    - Exemple : `float f = 3.14159; printf("%.2f", f);` // Affiche `3.14`

Combinaisons Utiles :

- **Alignement :** Vous pouvez utiliser des indicateurs pour l'alignement et l'espacement :
    - `%-10d` : Aligne un entier à gauche sur 10 espaces.
    - `%010d` : Remplit avec des zéros à gauche pour un total de 10 caractères.

Résumé des Spécificateurs Courants

Spécificateur	Description	Exemple
`%d`, `%i`	Entier signé	`printf("%d", 42);`
`%u`	Entier non signé	`printf("%u", 42);`
`%f`	Nombre flottant (float)	`printf("%f", 3.14);`
`%lf`	Nombre flottant (double)	`printf("%lf", 3.14);`
`%x`, `%X`	Entier hexadécimal	`printf("%x", 255);`
`%o`	Entier octal	`printf("%o", 255);`
`%c`	Caractère	`printf("%c", 'A');`
`%s`	Chaîne de caractères	`printf("%s", "Hello");`
`%p`	Adresse mémoire	`printf("%p", ptr);`
`%ld`	Entier long	`printf("%ld", 12345L);`

Spécificateur	Description	Exemple
`%lld`	Entier long long	`printf("%lld", 123456789LL);`
`%%`	Caractère pourcentage	`printf("%%");`